A Calculus Enabling Reuse and Composition of Component Assembly Specialization Processes

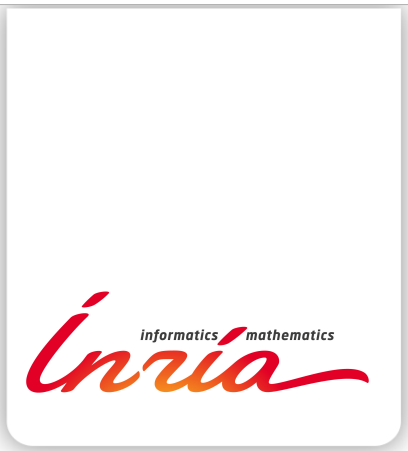
Vincent Lanore, Christian Pérez

A Calculus Enabling Reuse and Composition of Component Assembly Specialization Processes

Vincent Lanore, Christian Pérez

RESEARCH REPORT
N° 8761
July 2015
Project-Team Avalon

ISSN 0249-6399 ISRN INRIA/RR--8761--FR+ENG



A Calculus Enabling Reuse and Composition of Component Assembly Specialization Processes

Vincent Lanore, Christian Pérez

Project-Team Avalon

Research Report n° 8761 — July 2015 — 21 pages

Abstract: Software specialization exists in various forms in the literature ranging from product lines to adaptation of high-performance applications. In particular, specialization of component assemblies has been the focus of extensive research throughout the years and brings about specific challenges such as variant selection and hierarchy. We argue that many (possibly automatic) assembly specialization processes share a common structure. This paper presents a calculus which aims at providing a generic framework to ease reuse and composition of component assembly specialization processes. We show how this calculus can encode various features from the component model literature and discuss the existence of specialization processes in the literature and the usefulness of reusing and composing them.

Key-words: formal calculus, software specialization, component models, hierarchy, composition

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Un calcul permettant la réutilisation et la composition de processus de spécialisations d'assemblage

Résumé : La sépcialisation logicielle existe sous diverses formes dans la littérature allant des lignes de produits à l'adaptation d'applications haute performance. En particulier, la spécialisation d'assemblage de composants a fait l'objet de nombreux travaux et apporte des problématiques spécifiques comme la sélection de variantes et la gestion de la hiérarchie. De nombreux mécanismes de spécialisation d'assemblages, possiblement automatiques, se ressemblent. À partir de ce constat, nous proposons un calcul dont l'objectif et de faciliter la réutilisation et la composition de tels mécanismes (ou processus) de spécialisation. Nous montrons comment ce calcul permet d'exprimer, à travers des encodages, diverses fonctionnalités de la litérature composants. Nous discutons également de l'utilité de notre modèle, notamment à travers des exemples de processus de spécialisation.

Mots-clés : calcul formel, spécialisation logicielle, modèles à composants, hiérarchie, composition

Contents

A Calculus Enabling Reuse and Composition of Component Assembly Specialization Processes

Vincent Lanore, Christian Pérez

July 22, 2015

oftware specialization exists in various forms in the literature ranging from product lines to adaptation of high-performance applications. In particular, specialization of component assemblies has been the focus of extensive research throughout the years and brings about specific challenges such as variant selection and hierarchy. We argue that many (possibly automatic) assembly specialization processes share a common structure. This paper presents a calculus which aims at providing a generic framework to ease reuse and composition of component assembly specialization processes. We show how this calculus can encode various features from the component model literature and discuss the existence of specialization processes in the literature and the usefulness of reusing and composing them.

1 Introduction

Software specialization is the process of modifying a general-purpose program in order to make it more efficient for a specific subset of possible inputs or use cases. Examples of software specialization include partial compilation as well as application adaptation to specific hardware.

By extension, the process of building an application through successive implementation decisions can be seen as a form of specialization from an (abstract) application description to a (concrete) implementation. With this more general definition, processes such as variant selection or instantiation from a feature model are a form of software specialization.

In particular, specialization of component assemblies has been the focus of extensive research and raises specific challenges. Examples include component-based feature models and automatic variant selection in component models. Challenges raised include efficient exploration of the decision space and composite specialization in hierarchical models.

We argue that those component assembly specialization processes share a common structure: a sequence of specialization decisions are made until the assembly satisfies specific constraints. With this perspective, composition and reuse of such processes should be easy but the current lack of technological or formal conventions make it difficult.

So as to ease reuse and composition of component assembly specialization processes, we propose a formal specialization calculus for component assemblies. This calculus formalizes partial specialization in hierarchical component assemblies as an operation on a formal assembly. In this context, a specialization process is an oracle which chooses which operation to trigger from a set of possible specialization operations. Such oracles can easily be composed and reused. Generic oracles can even be written to implement generic decision tree exploration algorithms.

We evaluate the calculus following two directions. First, we consider several common component model features from the literature and show how to encode them in our approach. Second,

we discuss specialization processes from the literature, such as product lines and automatic variant selection. In addition, we show how those processes can be composed and we argue about the usefulness of such a composition.

The structure of this paper is as follows: Section 2 introduces the calculus through a series of formal definitions and a running example; then, Section 3 shows the generality of the approach by encoding common component model features, whereas Section 4 discusses the usefulness of the approach; finally, Section 5 concludes and give some perspectives.

2 A Calculus for Assembly Specialization

This section presents our proposal: a generic calculus for component assembly specialization. It is defined by an *assembly model*, a *type system*, and an *operational semantics*. Figure 1 presents how those three parts come together to perform assembly specialization. The idea behind the calculus is to be able to perform step-by-step specialization, one component at a time, using a specialization relation provided by a type system.

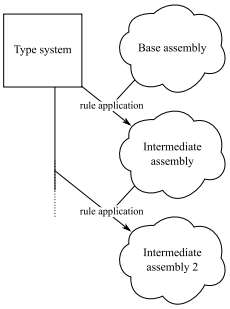


Figure 1: Overview of the use of the calculus. The type system is represented by a square while the cloud shapes represent component assemblies. The arrows represent the application of operations to transform the assembly step-by-step.

2.1 Assembly model

We introduce an assembly model whose goal is to be as general as possible. Our approach to generality is to propose a simple yet powerful model with few concepts and to introduce advanced concepts by encoding them in the assembly model. Examples of such encodings are given in Section 3.

The assembly model is a graph of *components* and *endpoints*. Components can represent either traditional components or be used to encode other architectural elements such as connectors. In that sense, component in the model can be thought of as generic architectural elements. In addition to that, *endpoints* are introduced; they aim to model interface constraints. Endpoints can be thought of as analogous to ports in classical component models.

To simplify the encoding of component interfaces (*i.e,* which outgoing edge corresponds to which endpoint), the definition is based on *multi-sorted list graphs* (as defined in [9]) instead of traditional graphs. List graphs provide *ordered edges* that have one source and several (ordered) targets. These ordered edges allow the encoding of interfaces comprising several endpoints without having to resort to edge labels or extra vertices.

Definition 1. *A* component assembly *is defined by:*

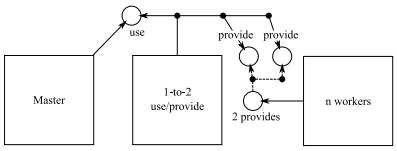


Figure 2: A half-specialized master/worker assembly. Components are represented with squares while endpoints are represented with circles. List edges are represented with lines and arrows similarly to [9]. Dotted list edges are implementations while solid list edges are interfaces.

- *a component set* C*;*
- *an endpoint set* E*;*
- *a connection set* X*;*
- *an endpoint implementation set* I*;*
- *an implementation source function* $i_s : I \to E$*;*
- *an implementation target function* $i_t : I \to E*$*;*
- *a connection source function* $x_s : X \to C$*;*
- *a connection target function* $x_t : X \to E*$*;*
- *a component type function* $t_c : C \to T_c$*;*
- *an endpoint type function* $t_e : E \to T_e$*.*

Where T_c and T_e are sets of component types and endpoint types respectively.

Figure 2 gives an example of a possible assembly and some hints at what endpoints are used for. The presented assembly is a partially-specialized master-worker assembly that would typically be a step during a specialization process. Note that the `1-to-2 use/provide` component is used to encode what would be a connector in a traditional component model (since our assembly model has no connector concept). Also note the endpoint hierarchy between the `1-to-2 use/provide` and `n workers` components. This hierarchy is here to serve as a buffer between two components which are not yet as specialized: the `1-to-2 use/provide` component is already specialized for the value 2 and it is connected to the adequate endpoints; the `n workers` component has not yet been specialized for the value 2 and it is still connected to the un-specialized root endpoint.

2.2 Type System

To perform specialization on black-box components, we propose to have a type system equipped with a specialization relation which specifies which component type can be specialized into which other. This section defines such a type system for the calculus. In addition to a straightforward type hierarchy, type systems introduce composite component types (in the traditional sense) and composite endpoint types (which are used to replace a single endpoint with multiple ones, *e.g,* when specializing for a specific arity).

Definition 2. *A* type system *is defined by the following:*

- *a set of component types* T_c*;*
- *a set of primitive component types* $T_c^p \subset T_c$*;*

- *a set of composite component types* $T_c^c \subset T_c$*;*
- *the signature constraint function* $s : T_c \to T_e*$*;*
- *a set of builders* $B \in transfo$*;*
- *a set of endpoint types* T_e*;*
- *a set of composite endpoints* $T_e^c \in T_e$*;*
- *a set of primitive endpoints* $T_e^p \in T_e$*;*
- *the component implementation function* $i_c : T_c^c \to B$*;*
- *the endpoint implementation function* $i_e : T_e^c \to P*$*;*
- *a specialization relation on component types* $<_c$ *(a partial order);*
- *a specialization relation on endpoint types* $<_e$ *(a partial order).*

Where $transfo$ is the set of transformations on assemblies (as defined canonically in [9, 13]). Section 2.3 gives constraints for these transformations and examples of such transformations are given in Figure 3 and explained below.

Component implementations (given by i_c) are meant to represent composites as traditionally understood in component models. The implementation of a composite component in the calculus is given by a *builder* which is a graph transformation which replaces the composite by its implementation in the system. Further details about the constraints on these transformations and how they are used are given in Section 2.4.

The implementation of an endpoint (given by i_e) is a list of endpoint types. Contrary to composite components, composite endpoints are not meant to be replaced by arbitrary assemblies but can only be divided into several endpoints. Endpoint implementation can typically be used to resolve n-to-m or 1-to-n connections by replacing a single composite endpoint with n endpoints; Figure 2 illustrates it with an endpoint *2-provides* being implemented by two distinct *provide* endpoints.

Components and endpoints which are neither primitive nor composite are called *abstract*. The sets of abstract components and endpoints are denoted T_c^a and T_e^a respectively. Those different sets of components and endpoints (abstract, composite and primitive) are meant to capture the level of abstraction of components and endpoints. Abstract component and endpoints are uninstantiable while primitive ones have a blackbox implementation and composite ones have an explicit model-level implementation (represented by i_c and i_e).

In addition, let us define primitive and abstract assemblies. This notion is useful for determining whether an assembly can be instantiated or not.

Definition 3. *An assembly is* primitive *iff*

$$\forall c \in C, t_c(c) \in T_c^p \quad and \quad \forall e \in E, t_e(e) \in T_e^p$$

2. *An assembly that is not primitive is* abstract.

Figure 3 presents a complete example of a type system for simple master-worker assemblies such as the one from Figure 2. At the middle-top is the hierarchy of connector component types which comprises various use/provide connectors; other component types include various masters, workers and worker sets; on the middle-right is the hierarchy of endpoints which comprises use and provide endpoints; at the bottom are the endpoint and component implementations.

Note that specialization arrows can represent various kinds of specializations: the arrow from *Worker* to *Worker A* corresponds to a variant selection while the arrow from *n identical workers*

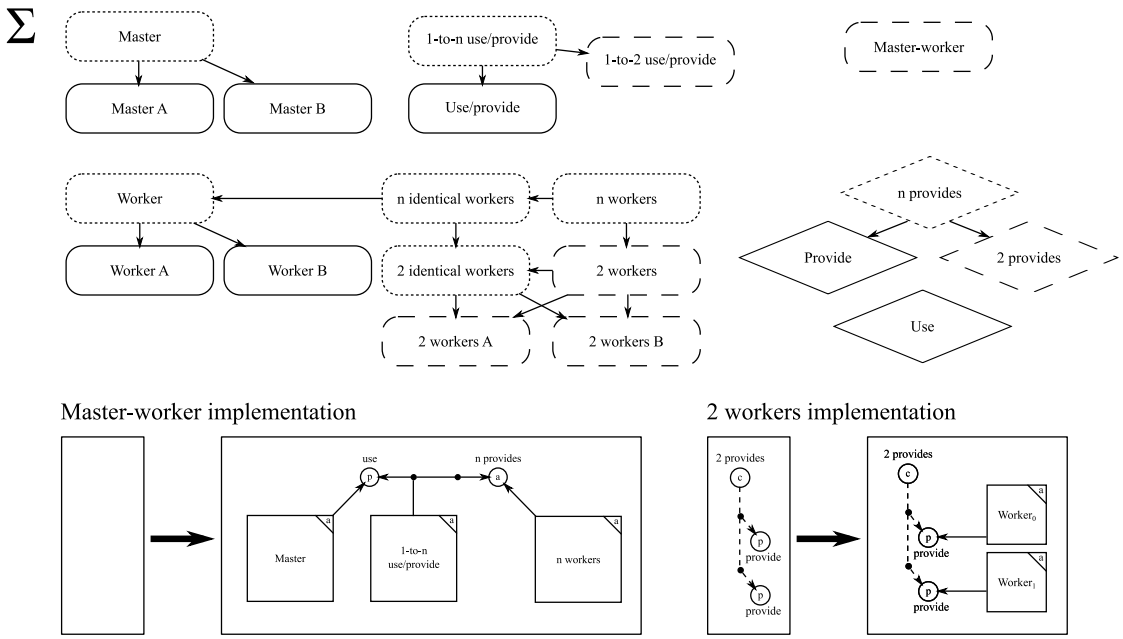


Figure 3: Example of a type system named Σ for master-worker assemblies. Component types are represented with rounded rectangles while endpoint types are represented with diamonds. Types with solid borders are primitive; those with dotted borders are abstract while those with dashed borders are composite. Arrows represent the specialization relation (if there is an arrow from A to B then B is a specialization of A). The full specialization relation is the transitive closure of the relation presented here. The boxes at the bottom represent the builders for the composite endpoints and components as graph transformation with implicit mapping. For clarity, components and endpoints in the transformations have their abstraction level written on them (a for abstract, c for composite or p for primitive)

to *2 identical workers* represent setting a value for an integer parameter and the arrow from *n workers* to *n identical workers* represents the addition of an architectural constraint.

Also note that, if we wanted to have all possible worker set sizes, this type system would need to include an infinite number of component types (*e.g,* one *i workers* type for every integer i) along with an infinite number of corresponding endpoints and implementations. A more practical way to specify such an infinite system is given in Section 3.2 in the form of a parametric type system.

2.3 Well-Formedness

Not all the assemblies and type systems as defined above make sense. For them to be meaningful, certain constraints must be respected. We model these constraint by introducing *well-formed assemblies*.

First, we introduce the notion of *component signatures*. A component signature in an assembly is the list of the types of the endpoints it is connected to.

Definition 4. *Let $c \in C$ be a component, if $\exists! i \in X, x_s(i) = c$ and $x_t(i) = \{i_0, \dots i_k\}$ then we define $sig(c) = \{t_e(i_0), \dots t_e(i_k)\}$ the* signature *of c.*

Definition 5. *The specialization relation on signatures $<_{sig}$ is defined by: let $r = \{r_0 \dots r_n\}$ and $s = \{s_0 \dots s_m\}$ be two signatures then $r <_{sig} s$ iff $n = m$ and $\forall i \in \{0 \dots n\}, r_i <_e s_i$.*

Theorem 1. *$<_{sig}$ is a partial order.*

Now, we can define what is a *well-formed* assembly.

Definition 6. *An assembly A is* well-formed *according to a type system Σ iff*

- *each component has at most one interface, i.e, $\forall c \in C, sig(c)$ is defined;*
- *the type of every endpoint and component in A appears in Σ;*
- *each component interface complies with the component's type signature constraint, i.e, $\forall c \in C, sig(c) <_{sig} s(t_c(c))$;*
- *composite implementations are well-formed, i.e, $\forall c \in T_c^c$ the origin of $i_c(c)$ is an assembly containing only the endpoints from $s(c)$ and the destination of $i_c(c)$ contains at least the endpoints from $s(c)$.*

The first condition means that all connections from a single component must be ordered. If multiple connections per components were allowed, there would be no order between the endpoints connected by different connections. This order on connected endpoints is important to encode which one is connected to which port of the component.

The third condition means that components cannot be connected to any kind of endpoints and that they are constrained by the type system *signature constraint*. This constraint is important as a guarantee on the connected endpoints which can be used for composite implementation.

The fourth condition means firstly that the only hypothesis a composite implementation is allowed to make is that the endpoints it is connected to respect the signature constraints and secondly that a composite implementation cannot remove the endpoints it is connected to.

In the rest of the paper, we assume that assemblies and type systems are well-formed unless specified otherwise.

2.4 Operational Semantics

This section defines operations that can specialize an assembly according to a type system. These operations implement local specialization (per component or per endpoint); they are meant to be used in multi-step transformations as presented in Figure 1.

Operations

- Specialize endpoint ($\mathbf{sp_e}(\mathbf{e}, \mathbf{B})$): let e be an endpoint of type A and $B \in T_e$ such that $B <_e A$ then replace type of e by B.
- Specialize component ($\mathbf{sp_c}(\mathbf{c}, \mathbf{B})$): let c be an component of type A and $B \in T_c$ such that $B <_c A$ and such that $sig(c) <_{sig} s(B)$ then replace type of c by B.
- Remove unused endpoint ($\mathbf{rm}(\mathbf{e})$): let e be an endpoint such that there exists no $i \in X$ such that $e \in x_t(i)$ and there exists no $i \in I$ such that $e \in i_t(i)$ then remove e from assembly and remove its implementation if it had one.
- Implement composite ($\mathbf{im_c}(\mathbf{c})$): let c be a composite component, then apply $i_c(c)$ to the subassembly composed of c, its outgoing connection if there is one and all the endpoints it is connected to.

- Implement endpoint ($\mathbf{im_e(e)}$): let e be a composite endpoint, add a new implementation i such that $i_s(i) = e$; create new endpoints $e_0 \dots e_n$ such that $n = |i_e(e)|$, $\forall i \in \{1 \dots n\} t_e(e_i) = i_e(e)_i$ and $i_t(i) = \{e_0 \dots e_n\}$.

Definition 7. *The* set of valid operations *on assembly A according to type system Σ is the set of all operations o, denoted $V(A, \Sigma)$, which verify:*

- *The target endpoint or component belongs to A.*
- *If o is a specialization operation ($\mathbf{sp_e}$ or $\mathbf{sp_c}$) then the target type is from Σ.*
- *All conditions for the operation are met (e.g, the target type for specialization is indeed a specialization of the current type).*

Definition 8. *A* specialization process *is an algorithm which takes as parameter a type system Σ and an assembly A and returns an operation from $V(A, \Sigma)$.*

This definition means that a specialization process is a black box responsible for selecting a valid operation at each step of the specialization. It can be thought of as an oracle or a chooser which selects a path in the tree of possible specialization decisions. Whether such specialization processes exist in the literature and whether the calculus presented here is useful to reuse and compose them are questions discussed in Section 4.

Theorem 2. *The five operations preserve well-formedness.*

In order of appearance in the well-formedness definition (see Section 2.3): the well-formedness of interfaces holds since no operation adds interfaces to an existing component; the new types all belong to the type system (constraint of sp_c and sp_e); the signature constraints are preserved because it is a condition for im_c and the builders are still well-formed because operations do not touch the type system.

2.5 Full Example

Figure 4 presents a full example of an assembly specialization from a very abstract single component assembly to a primitive multi-component assembly. Specifically, it deals with the example of a master-worker assembly built from the type system presented in Figure 3.

The figure presents the assembly at each step of the transformation, the set of valid operations at each step and the chosen operation(s) at each step. Not all specialization steps are presented in order to save space.

At first (assembly A_0), there is only a single *Master-Worker* composite component. So far no architectural decision has been taken apart from the fact that is assembly must be a master-worker style assembly. The first operation is to implement the *Master-Worker* component by applying the builder from the type system; this results in a assembly with a *Master* component and an abstract set of workers which are connected with an abstract $1 - to - n$ *use* connection (assembly A_1). Then, it is decided that the number of workers will be 2 and all the relevant endpoints and components are specialized for this value (the connection, the worker set and the multiple provide endpoint) resulting in assembly A_2; note that the endpoint specialization must be performed first so as to respect the signature constraints at all steps. None of the two composite components can be instantiated yet because they are connected to an uninstantiated composite endpoint. The composite endpoint in question is then instantiated resulting in assembly A_3. All the conditions are now met to instantiate the worker set which results in assembly A_4 (which resembles closely the example from Figure 2); note that the implementation of the

endpoint is used to connect the two workers directly to *Provide* endpoints. The composite connection is then instantiated in the same fashion resulting in assembly A_5. There is now an unused endpoint which is promptly removed with the appropriate operation resulting in assembly A_6; note that the two *Provide* endpoints are now fully independent. A final specialization step is taken to choose a variant for every master and worker component resulting in assembly A_7 which is primitive.

One can notice how hierarchy is handled: the hierarchy of endpoint implementations serves as a buffer between two composites which will each require several endpoints to instantiate. Instead of requiring a simultaneous specialization or instantiation of both components, the calculus permits to specialize and instantiate each one in turn.

A second remark is that two meaningful architectural decisions were made: first, the value of n was set (between A_1 and A_2) and second, the variants for all components were chosen (between A_6 and A_7). Also note that, at every step most valid operations correspond either to a mandatory step in resolving a previous meaningful decision (*e.g,* instantiating a composite) or to another meaningful decision (*e.g,* specializing to *2 identical workers* at A_4 would have put an additional architectural constraint).

A last point to note is that, in this specific example, there is no way to make an inconsistent decision (because of the signature constraints); a process choosing operations randomly in the valid operations set would always produce a primitive master-worker assembly. While this is not true in the general case, it shows that signature constraints can efficiently forbid invalid cases.

2.6 Calculus Variant: Reversible Operations

In the calculus as presented so far, not all operations are reversible. Indeed, $\mathbf{im_c}$ and **rm** remove elements from the assembly with no way to revert unless extra information is available. This limitation can be problematic for decision tree traversal. In order to remove this limitation, this section presents alternative operations which preserve the structure during specialization, providing efficient reverse operations at the cost of extra unremovable nodes in the assembly.

The idea behind this calculus variant is to maintain a component hierarchy similar to the endpoint hierarchy and to forbid removal of endpoints and nodes. To this end, we need to extend the assembly model to allow component-to-component implementation edges and to modify the operations to maintain the implementation hierarchy.

Assembly Model Modification. To let implementation edges connect components, the definitions of i_s and i_t are modified as follows:

- $i_s : I \to (E \cup C)$;
- $i_t : I \to (E* \cup C*)$.

In addition, for an assembly to be well-formed, an implementation must now have a component as a source iff it has components as targets.

Reversible Operations.

- $\mathbf{sp_e}$, $\mathbf{sp_c}$ and $\mathbf{im_e}$ are unchanged.
- **rm** no longer exists.
- $\mathbf{im_c}(\mathbf{c})$ is not any longer allowed to remove the original component and, in addition, it creates a new implementation i whose source is c and whose targets are all the components created by the builder.

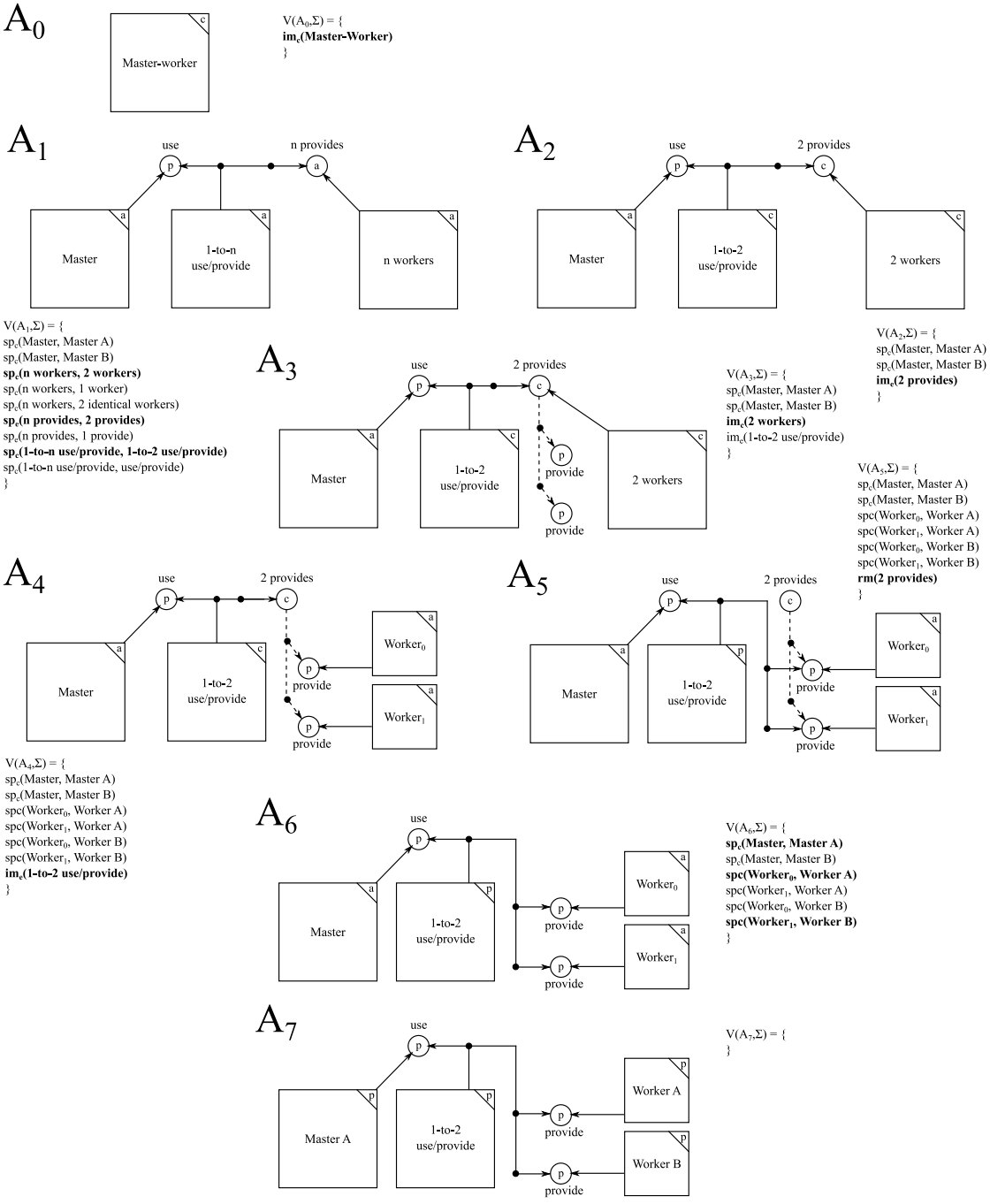


Figure 4: Full example of the specialization of an abstract master-worker assembly to a primitive assembly. Consecutive steps in the specialization are presented in a top-down left-right order. For each step, the list of possible operations is presented and the operation(s) which have been chosen are highlighted in bold. Conventions for assemblies are the same as for Figure 2 and Figure 3.

Now that the calculus keeps the hierarchy structure during specialization, we need to define what additional data must be kept and how to reverse operations.

Additional Data. For operations to be reversible, the following data must be kept during specialization:

- the list of operations made so far with their parameters;
- for specialization operations (*i.e,* $\mathbf{sp_c}$ and $\mathbf{sp_e}$), the type of the endpoint or component *before* the specialization (*e.g,*, if specialized from type A to type B then the operation is $\mathbf{sp}_*(*, B)$ and A must be stored in addition).

Reverse Operations. At any time, the last operation that has been done can be reversed in the following way:

- if it was a $\mathbf{sp_e}$ or $\mathbf{sp_c}$ operation, then change the endpoint or component type back (all the necessary information is stored);
- if it was a $\mathbf{im_e}(\mathbf{x})$ or $\mathbf{im_c}(\mathbf{x})$ operation, then, considering $\exists i \in I, i_s(i) = x$, remove all components/endpoints in $i_t(i)$ and remove i.

These new reversible operations have the following costs: the list of operations must be stored with extra type information ($O(\#operations)$ in extra space), the hierarchy of composite components must be kept during specialization ($O(\#components \times maxCompositeDepth)$ in terms of extra components and extra connections), and the time cost of reversing is $O(1)$ plus the actual modification of the assembly which is unavoidable.

In addition to providing easily reversible operations, this calculus variant keeps additional structural data regarding which component was instantiated by which composite. This can be useful to model nested composites (see Section 3.1 for details).

3 Generality of the Model

The first step in evaluating the calculus is to estimate how general it is, that is how many of the existing component models it can itself model. Benefits of being general include easy reuse across component models and a wide area of application. For the calculus and the assembly model in particular to be general, commonly found features of component models must be easy to encode. Examples of such features include hierarchy, connectors, and and genericity. An example on how to encode connectors was present in the running master-worker example from Section 2.

Finding an encoding for a new concept into the assembly model is easy but finding a good one is not. A good encoding is an encoding which does not increase dramatically the complexity of the operations on the encoded objects. In our particular case, a good encoding of a new concept or feature is one in which a meaningful architectural decision is easy to implement (in terms of number of operations and ratio of valid paths in the decision tree).

3.1 Hierarchy

Hierarchy is a commonly found feature in academic component models which allows to implement a component with a component assembly. Examples of models with hierarchy include [7, 8, 1, 4]. Depending on the model, hierarchy can be either just a efficient way to describe subassemblies in an otherwise flat model or hierarchy can be present in the assembly in the form of nested components.

The proposed calculus is capable of modelling both. The basic calculus features composite components with an implementation which can be used to replace the component by its assembly implementation. In this case the assembly stays flat at all time (since a composite is destroyed when replaced by its implementation).

The calculus variant with reversible operations (see Section 2.6) is capable of modelling nested components. Indeed, in this variant of the calculus, composite components are not destroyed when implemented. They are connected to their implementation whose components can in turn be connected to their own implementations and so on, creating a tree structure. This tree structure can be thought of as a topograph (as in bigraphs [12]) and encodes exactly the nesting of composite components.

The operations and type systems have been made specifically with hierarchy in mind and are thus easy to use with it. One typical problem encountered in hierarchical models which is solved by this approach is the step-by-step specialization of two abstract components connected by an abstract connection (*e.g,* two generic components connected with a n-to-m connector) which is handled here using endpoints as a buffer.

3.2 Genericity

Component model literature has proposed genericity [3] to ease reuse of components by having components parametrized by types or values.

In the case of the calculus, genericity (from a specialization process's perspective) can be implemented in a type system by providing one type per possible parameter value. Specialization operations such as setting an integer parameter must be encoded in type systems using an infinite number of types (as illustrated in Figure 3) which is impractical both for specifying the type system and enumerating possible operations at a given step in a transformation.

This section defines parametrized type systems which ease those two tasks by providing a model to describe an infinite type system in a finite way. Along with the type system definition, a mapping to a (non-parametric) type system is also provided.

Definitions Let us first define a simple parameter grammar. Parameters can either be endpoint/parameter types or values. A value can be an actual value (*e.g,* an integer), a value type (*e.g,* Integer), or it can be undefined.

Definition 9. *Parameters are defined by:*

$$P_t \ ::= \ TYPE$$

$$P_v \ ::= \ UNDEF \mid VTYPE \mid VVALUE$$

Where $TYPE$ is a component or endpoint type (*i.e,* from $T_c \cup T_e$), $UNDEF$ is a constant which denotes that nothing is defined, $VTYPE$ is a value type from a set of value types (*e.g,* integer, string, float) and $VVALUE$ is a value (*e.g,* 3, "*hello*", or 5.2).

Let us now define parametric component and endpoint types. The constraint, build, implementation and abstraction functions are just straight ports from non-parametric type systems while the two parameter lists are new.

Definition 10. *A* parametric component type *is defined by:*

- *a list* $p_t \in P_t*$ *of type parameters;*
- *a list* $p_v \in P_v*$ *of value parameters;*
- *a constraint function* $c : p_t \cup p_v \to P*$*;*

- *a build function* $b : p_t \cup p_v \cup transfo$*;*
- *an abstraction function* $a : p_t \cup p_v \rightarrow \{primitive, composite, abstract\}$*.*

Definition 11. *A* parametric endpoint type *is defined by:*

- *a list* $p_t \in P_t*$ *of type parameters;*
- *a list* $p_v \in P_v*$ *of value parameters;*
- *a implementation function* $i : p_t \cup p_v \cup P*$*;*
- *an abstraction function* $a : p_t \cup p_v \rightarrow \{primitive, composite, abstract\}$*.*

We can now define a parametric type system (PTS). It is an extension of a non-parametric type system (NPTS) where some abstract component and endpoint types are associated to parametric types.

Definition 12. *A* parametric type system *(PTS) is defined by:*

- *a non-parametric type system* S*;*
- *the parametric component subset* $T_c^{par} \subset T_c^a$*;*
- *the parametric endpoint subset* $T_e^{par} \subset T_e^a$*;*
- *a set of parametric component types* P_c^{par}*;*
- *a set of parametric endpoint types* P_e^{par}*;*
- *the parametric component function* $p_c : T_c^{par} \rightarrow P_c^{par}$*;*
- *the parametric endpoint function* $p_e : T_e^{par} \rightarrow P_e^{par}$*.*

With this approach, each parametric type models a whole family of non-parametric types. Having such a family be a specialization of a non-parametric type makes sense but the reverse is not true. For this reason, we define well-formed PTSes to forbid it. We assume in the remaining of the section that PTSes are well-formed.

Definition 13. *A PTS is said to be* well-formed *iff*

- S *is well-formed;*
- $\forall t \in T_c^{par}, t$ *is a minimum for* $<_c$*;*
- $\forall t \in T_e^{par}, t$ *is a minimum for* $<_e$*.*

Being a minimum for the specialization relation means that there can be no type which is more specialized, thus forbidding the case we wanted to forbid.

We also define a specialization relation on parametric types. This relation is analogous to the specialization relation on NPTSes.

Definition 14. *The specialization relation on value parameters is defined by: a* $VVALUE$ *is a specialization of the corresponding* $VTYPE$ *which is a specialization of* $UNDEF$*.*

2. *The specialization relation on parametric types is defined by: two parametric types* A *and* B *verify* $A <_{par} B$ *iff their parameter lists have identical lengths and* A*'s parameters are specializations of* B*'s corresponding parameters.*

Theorem 3. $<_{par}$ *is a partial order.*

We can now define the NPTS generated by a PTS. Basically, it adds to the type system all the possible variations of the parametric types and updates everything necessary. To save space, we do not give a fully explicit definition but there is nothing complex or unexpected anyway.

Definition 15. *The* generated non-parametric type system *of the parametric system*

$$\Sigma = (S, T_c^{par}, T_e^{par}, P_c^{par}, P_e^{par}, p_c, p_e)$$

with

$$S = (T_c, T_c^p, T_c^c, s, B, T_e, T_e^c, T_e^p, i_c, i_e, <_c, <_e)$$

is defined by the following:

- T_c *and* T_e *are respectively augmented with* P_c^{par} *and* P_e^{par} *and all types* t *such that* $\exists t' \in P_e^{par} \cup P_c^{par}, t <_{par} t'$*;*
- $<_c$ *is augmented with* $<_{par}$*,* p_c *and* p_e*;*
- T_c^p*,* T_e^p*,* T_c^c *and* T_e^c *are updated according to the a function of the newly added component and endpoint types;*
- s *is updated according to the c function of newly added component types;*
- B *and* i_c *are updated according to the b function of newly added component types;*
- i_e *is updated according to the i function of newly added endpoint types.*

Discussion and Example Let us have a brief discussion about the genericity encoding described above.

First, Figure 5 shows the example of a *n workers of type T* component similar the the one found in Figure 3. The constraint, build and abstraction functions are described using ad-hoc pseudocode and a graphical representation of the build function is presented. This example shows that a simple parametric component is fairly easy to describe and that the encoding is usable in practice.

Second, parametric type systems can be used to ease the operation selection process. Indeed, compared to a non-parametric type system, a parametric one provides additional structure which can be used to sort possible operations in a meaningful way. For example, the specialization of a component of type *n workers of type A* can be done either by setting the value of n or by specializing T. In a non-parametric type system these two sorts of decisions are indistinguishable and correspond to an infinity of types while the structure is apparent in a parametric type system.

4 Specialization Processes

While the previous section has discussed the generality of the approach with regards to component model features, this section deals with specialization processes: what sorts of specialization processes exist in the literature and what does our approach allow in terms of reuse and composition?

4.1 Specialization Processes in The Literature

Several component models from the literature propose some form of assembly generation or assembly-level assembly optimization. Examples of domains where such approaches exist include scientific computing and cloud computing where performance and scalability are important focuses.

n identical workers

p_t = {Worker}

p_v = {INTEGER}

```
c: function c(Worker_type t, Integer n)
       if n is a value then
           return [endpoint_type(n+" provides")]
       else
           return [endpoint_type("n provides")]
```

```
a: function a(Worker_type t, Integer n)
       if n is a value then
           return COMPOSITE
       else
           return ABSTRACT
```

```
b: function b(Worker_type t, Integer n)
       if n is a value then
           for i in {1... n} do
               tmp_c = new Component(t)
               tmp_e = sig_constraint.implem[i]
               connect(tmp_c, tmp_e)
       else
           return NO_BUILDER
```

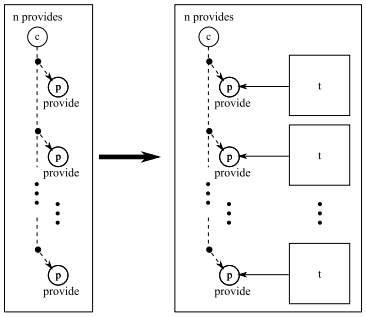


Figure 5: Example of parametric system for a worker set. Functions are described using pseudocode and the builder function is also represented as a pseudo graph transformation for maximum clarity.

A first, simple form of automatic assembly optimization is *variant selection*. Variant selection consists in choosing, for each component in a provided assembly, one implementation or algorithm (called a variant of that component) so as to optimize a non-functional property such as performance of the assembly for a specific hardware. An example of a component model which propose automatic variant selection is PEPPHER [10, 2]. Variant selection can be modelled in the calculus presented here as a specialization from an abstract component to a primitive one (the variant).

Other component models have a more general approach and propose assembly-level optimization (which can modify the assembly, as opposed to variant selection which only selects variant for already connected components). Examples of such approaches include BIP optimizations [5, 6]. The possible natures of optimizations in such a context may include changing the communication topology, selecting variants, deciding the size of component collections, merging components or placing the components on resources. Some of these forms of optimization can be easily modelled by the calculus proposed here as they can be seen as local specializations (*e.g,* variant selection, size of collections) while others might require more work as they are not local (*e.g,* merging components) or not strictly speaking specializations (*e.g,* placement on resources). In the case where all optimizations can be modelled with our approach, the algorithm which chooses which optimization to perform can be made into a specialization process.

The example from the literature which resembles our approach the most is HLCM [4]. HLCM is a connector-based generic hierarchical component model which proposes to generate a low-level flat component assembly from a high-level abstract assembly through an automatic generation process. This approach matches exactly the calculus –which was actually one of the motives behind this calculus– and requires the two extensions proposed here in order to be fully modelled (reversible operations and genericity). The default HLCM implementation implements a default chooser function which lists all the possible operations at a given step and chooses the first one. This default implementation (or alternative user-provided choosers) constitute specialization processes in the sense of our approach.

Another area of software engineering where specialization processes exist is the study of product lines and feature models. This domain is concerned with modelling families of program using features as parameters. Some works in this domain deal with automatic feature selection with respect to a set of constraints and some of those works describe programs with component assemblies. Examples of such works include [14, 15]. Feature selection can be modelled in our approach using parametric type systems (see Section 3.2). While some global constraints might be complex to enforce, algorithms for automatic feature selection are good examples of automatic specialization processes.

4.2 Generic Processes

A first advantage of our approach is to allow reuse of specialization processes across applications. While some specialization processes might be too application-specific to be reused, some other processes might be generic or general enough to be used in a variety of contexts.

One example of such a generic process is the default HLCM implementation [4]. While this is only a default implementation and meant to be replaced by user-defined choosers, it can also be seen as a generic specialization process, *i.e,* a specialization process which can work with any type system and any assembly, although it does not guarantee termination.

As we have seen in the full master-worker specialization example (see Section 2.5) a lot of meaningful architectural constraints can be captured by signature constraints. If signature constraints are strong enough to guarantee that a primitive assembly fits the user's purposes, then the challenge is to find a sequence of operations which results in a primitive assembly. There is a rich literature in exploration of decision trees which could be used directly to make generic specialization processes which guarantee that they will produce a primitive assembly if possible. Such a generic process could be reused in a large variety of contexts.

4.3 Composition and Reuse

As we have seen in the previous sections, our approach can model both actual decision processes such as assembly optimization and generic subassemblies (as parametrized types). Combining those two types of processes would make sense and be useful, for example to have decision processes which make use of generic subassemblies.

Our approach allows to mix several specialization processes for a single assembly through two types of composition: spatial and temporal. While we do not define these two composition types extensively, they are simple enough that we can provide here a simple explanation of how and why they work.

Spatial Composition Spatial composition of specialization processes can be done by restraining processes to subassemblies. The definition of a specialization process works without problem with a subassembly with the only consequence being that the set of possible transformations at a given step will be reduced.

One possible use of spatial composition is to assign processes to generic subassemblies in order to specialize them until they are primitive. Another possible use is to separate processes in a hierarchical fashion with subprocesses responsible for specializing specific parts of the whole assembly.

Note that, because of the local nature of the operations from the calculus, several specialization processes which have been composed spatially can be run in parallel with very little synchronization.

Temporal and Spatio-Temporal Composition Specialization processes can also be composed temporally using mechanisms such as workflow-style dependencies. Useful examples in-

clude forcing the specialization of a subassembly to finish before running a higher-level process or prioritizing processes, *e.g,* to ensure termination.

Overall, the composition of specialization processes, while still lacking some formalization (*e.g,*, well-defined interfaces between processes and assemblies), open new possibilities of high-level specification of specialization processes.

5 Conclusion

We have presented a calculus for step-by-step specialization of component assemblies. Specifically, we have presented a component assembly model, a type system definition and a set of operations which allow local specialization. We have also introduced specialization processes which perform step-by-step specialization of assemblies until a specific condition is met.

We have seen in Section 3 that the calculus proposed in this paper is capable of encoding common component model features from the literature in a way that is not too complex. This means that our approach is compatible with most existing component models out there.

In Section 4, we have discussed the existence of specialization processes in the literature and the usefulness of reusing and composing them. We have determined that although certain limitations were encountered, such as the difficulty to enforce global constraints, most of the examples we considered can be made into specialization processes. Moreover, the possibility to compose specialization processes and the possibility to have generic specialization processes open up interesting possibilities.

Perspectives for this work include implementation and application to a concrete case. Such a study would allow a quantitative evaluation of the reuse capability and expressive power of the approach. A good candidate to serve as a base for an implementation of our approach would be HLCM as it already implements similar mechanisms.

Another possible perspective is a deeper study of the possibility offered by generic specialization processes. There is a rich literature about exploration of decision trees but none, to our knowledge, about the specific case of automatic generation of hierarchical component models. In particular, the automatic optimization of non-functional properties such as performance is an ongoing challenge in the HPC and component models communities.

Finally, it would be interesting to propose encodings for some component model features which are not considered in this paper. Examples of such features include assembly transformations and resource models such as, for example, what can be found in DirectMOD [11]. A common issue with these models is that transformations are highly dependent of factors such as number/type of ports and resource allocation which may be encountered in various configurations. Automatic generation of transformations using our approach could simplify greatly transformation specification in such models.

References

[1] Ananda Basu, Marius Bozga, and Joseph Sifakis. Modeling Heterogeneous Real-time Components in BIP. In *Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods*, SEFM '06, page 3–12, Washington, DC, USA, 2006. IEEE Computer Society.

[2] Siegfried Benkner, Sabri Pllana, Jesper Larsson Träf, Philippas Tsigas, Uwe Dolinsky, Cedric Augonnet, Beverly Bachmayer, Christoph Kessler, David Moloney, and Vitaly Osipov. Pep-

pher: Efficient and productive usage of hybrid computing systems. *IEEE Micro*, 31(5):28–41, 2011.

[3] Julien Bigot and Christian Pérez. Increasing Reuse in Component Models through Genericity. In Stephen H. Edwards and Gregory Kulczycki, editors, *Proceedings of the 11th International Conference on Software Reuse, ICSR '09*, Formal Foundations of Reuse and Domain Engineering, pages 21–30, Falls Church, VA, United States, September 2009. Springer-Verlag.

[4] Julien Bigot and Christian Pérez. High Performance Composition Operators in Component Models. In Ian Foster, Wolfgang Gentzsch, Lucio Grandinetti, Gerhard R. Joubert, editor, *High Performance Computing: From Grids and Clouds to Exascale*, volume 20 of *Advances in Parallel Computing*, page 182–201. IOS Press, 2011.

[5] Borzoo Bonakdarpour, Marius Bozga, Mohamad Jaber, Jean Quilbeuf, and Joseph Sifakis. From high-level component-based models to distributed implementations. In *Proceedings of the tenth ACM international conference on Embedded software*, EMSOFT '10, page 209–218, New York, NY, USA, 2010. ACM.

[6] M. Bozga, M. Jaber, and J. Sifakis. Source-to-Source Architecture Transformation for Performance Optimization in BIP. *Industrial Informatics, IEEE Transactions on*, 6(4):708–718, Nov 2010.

[7] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The fractal component model and its support in java. *Software: Practice and Experience*, 36(11-12):1257–1284, 2006.

[8] T. Bures, P. Hnetynka, and F. Plasil. SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model. In *Software Engineering Research, Management and Applications, 2006. Fourth International Conference on*, page 40–48, Aug 2006.

[9] Maarten de Mol and Arend Rensink. On a graph formalism for ordered edges. *Electronic Communications of the EASST*, 29, 2010.

[10] Christoph Kessler and Welf Löwe. Optimized composition of performance-aware parallel components. *Concurrency and Computation: Practice and Experience*, 24(5):481–498, 2012.

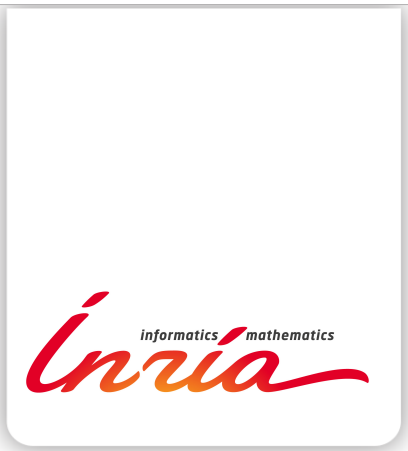
[11] Vincent Lanore and Christian Perez. A Reconfigurable Component Model for HPC. In *The 18th International ACM Sigsoft Symposium on Component-Based Software Engineering (CBSE'2015)*. ACM, 2015.

[12] Robin Milner. Bigraphical reactive systems: basic theory. Technical report, Technical Report 523, Computer Laboratory, University of Cambridge, 2001.

[13] Ulrike Prange and Hartmut Ehrig. Graph transformation in adhesive hlr categories. In *Pfalzgraf, J.(Hrsg.): Advances in Multiagent Systems, Robotics and Cybernetics: Theory and Practice. Proceedings of Intern. Conf. on Systems Research, Informatics and Cybernetics*. Citeseer, 2005.

[14] Marko Rosenmüller and Norbert Siegmund. Automating the configuration of multi software product lines. *VaMoS*, 10:123–130, 2010.

[15] Jules White, Brian Dougherty, Doulas C Schmidt, and David Benavides. Automated reasoning for multi-step feature model configuration problems. In *Proceedings of the 13th International Software Product Line Conference*, pages 11–20. Carnegie Mellon University, 2009.



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399